Florian Pudlitz, Andreas Vogelsang, Florian Brokhausen

A Lightweight Multilevel Markup Language for Connecting Software Requirements and Simulations

The final authenticated version is available online at https://doi.org/10.1007/978-3-030-15538-4_11

Pudlitz, F., Vogelsang, A., & Brokhausen, F. (2019). A Lightweight Multilevel Markup Language for Connecting Software Requirements and Simulations. In Structured Object-Oriented Formal Language and Method (pp. 151–166). Springer International Publishing. https://doi.org/10.1007/978-3-030-15538-4_11

A Lightweight Multilevel Markup Language for Connecting Software Requirements and Simulations

Florian Pudlitz[0000−0002−0006−1853], Andreas Vogelsang[0000−0003−1041−0815], and Florian Brokhausen

Technische Universität Berlin, Germany
{florian.pudlitz,andreas.vogelsang}@tu-berlin.de,
florian.brokhausen@campus.tu-berlin.de

Abstract. [**Context**] Simulation is a powerful tool to validate specified requirements especially for complex systems that constantly monitor and react to characteristics of their environment. The simulators for such systems are complex themselves as they simulate multiple actors with multiple interacting functions in a number of different scenarios. To validate requirements in such simulations, the requirements must be related to the simulation runs. [**Problem**] In practice, engineers are reluctant to state their requirements in terms of structured languages or models that would allow for a straightforward relation of requirements to simulation runs. Instead, the requirements are expressed as unstructured natural language text that is hard to assess in a set of complex simulation runs. Therefore, the feedback loop between requirements and simulation is very long or non-existent at all. [**Principal idea**] We aim to close the gap between requirements specifications and simulation by proposing a lightweight markup language for requirements. Our markup language provides a set of annotations on different levels that can be applied to natural language requirements. The annotations are mapped to simulation events. As a result, meaningful information from a set of simulation runs is shown directly in the requirements specification. [**Contribution**] Instead of forcing the engineer to write requirements in a specific way just for the purpose of relating them to a simulator, the markup language allows annotating the already specified requirements up to a level that is interesting for the engineer. We evaluate our approach by analyzing 8 original requirements of an automotive system in a set of 100 simulation runs.

Keywords: Markup language · requirements modeling · simulation · test evaluation.

1 Introduction

In many areas, software systems are becoming increasingly complex through the use of open systems, highly automated or networked devices. The complexity

leads to an increasing number of requirements, which are often expressed in natural language [9]. To master the complexity of development and test management, simulation is increasingly being used to anticipate system behavior in complex environments. Simulation has several advantages over classic testing. Tests only pass or fail, but there is little information about the contextual situation. Additionally, simulations are more flexible towards covering variations in context behavior.

However, in current practice and especially in large companies, simulation and requirements activities are often not aligned. Simulation scenarios are not derived from requirements but handcrafted by specialized simulation engineers based on their own understanding of the problem domain. On the other hand, the results of simulation runs are not fed back to the level of requirements, which means that a requirements engineer does not benefit from the insights gained by running the simulation. This misalignment has several reasons. First, requirements engineering and simulation is often conducted in different departments. Second, simulators are complex systems that need to be configured by simulation experts. That makes it hard for requirements engineers to use simulators. Third, requirements and simulations are on different levels of abstraction which makes it hard to connect events generated by the simulation to requirements, especially, when they are written in natural language. As a result, the simulation scenarios are often unrealistic and do not ensure that all requirements are covered.

Modeling can help closing this gap between requirements and simulation. However, if the necessary models are too formal, requirements engineers fear the effort to model the requirements. Therefore, we propose a lightweight modeling approach that allows engineers to annotate their natural language requirements instead of expressing them as models. Based on these annotations, the respective part of a requirement can be linked to a simulation event. By analyzing logs of simulation runs for the linked simulation events, we can feed back information about system execution to the level of the annotations and thereby to the level of requirements. The available annotations build a markup language. A distinct feature of our markup language is that it contains annotations on different levels of detail. An engineer can decide how detailed he or she wants to annotate a requirement. The more detailed a requirement is annotated, the more information can be retrieved from a simulation run.

In this paper, we present the general idea of our approach, the details of the markup language, and an evaluation on a Cornering Light System. Our approach provides a minimal invasive way to connect (existing) requirements with simulation. Thereby, requirements engineers can profit from insights gained by simulation much faster and without having to invest in extensive modeling efforts. The requirements engineer gets feedback whether the requirements are covered by the current selection of simulation scenarios and whether there are misconceptions in the requirements that are uncovered by the simulation (e.g. false assumptions).

2 Background and Related Work

Testing and Simulation: Software Testing is the verification that a software product provides the expected behavior, as specified in its requirements. The conventional development and testing process for complex systems is based on the V-model, which structures the development process into phases of decomposition of the system elements and their subsequent integration. Each requirement being specified on a certain level of abstraction is reflected by a test case on the same level which determines whether the requirement has been implemented correctly. The increasing complexity of the systems, the many possible test cases, and the uncertainty about the system's context challenge this conventional testing process. Therefore, the use of simulations is becoming more and more popular.

Simulation is the imitation of the operation of a real-world process or system [1]. The act of simulating something first requires that a model is developed; this model incorporates the key characteristics, behavior, and functions of the selected physical or abstract system or process. A simulator is a program that is able to run a simulation. Each simulation run is one execution of the simulation.

When simulation is used in a systems development process, the model usually consists of a submodel that describes the system-under-development (SuD) and one or several submodels that describe the operational environment of the SuD. The simulation represents the operation of the SuD within its operational context over time.

A simulation scenario defines the initial characteristics and preliminaries of a simulation run and spans a certain amount of time. The scenario defines the global parameters of the operational context model. The model of the SuD is not affected by the definition of the simulation scenario. Therefore, a simulation scenario can be compared to a test case in a conventional testing processes. The expectation is that the SuD performs according to its desired behavior in a set of representative simulation scenarios.

Requirements and Test Alignment: Alignment of requirements and test cases is a well-established field of research and several solutions exist. Barmi et al. [2] found that most studies of the subject were on model-based testing including a variety of formal methods for describing requirements with models or languages. In model based testing, informal requirements of the system are the base for developing a test model which is a behavioral model of the system. This test model is used to automatically generate test cases. One problem in this area is that the generated tests from the model cannot be executed directly against an implementation under test because they are on different levels of abstraction. Additionally, the formal representation of requirements often results in difficulties both in requiring special competence to produce [10], but also for non-specialist (e.g. business people) in understanding the requirements. The generation of test cases directly from the requirements implicitly links the two without any need for manually creating (or maintaining) traces [3]. However, depending on the level of abstraction of the model and the generated test cases, the value of the traces might vary. For example, for use cases and system test

cases, the tracing was reported as being more natural in comparison to using state machines [5]. Errors in the models are an additional issue to consider when applying model-based testing [5].

Lightweight Requirements Modeling: The use of constrained natural language is an approach to create requirements models while keeping the appearance of natural language. Several authors propose different sets of sentence patterns that should be used to formulate requirements [8, 4]. Besides the advantage that requirements are uniformly formulated, the requirements patterns enrich parts of the requirement with information about the semantics. This information can be used to extract information from the requirements. Lucassen et al., for example, use the structure of user stories to automatically derive conceptual models of the domain [7]. With our approach, we try to combine the strength of lightweight requirements annotations with the potential to be enriched with behavioral information collected in simulations.

End-to-end Tooling: A comparable approach to validate requirements within a testing and simulation environment in an end-to-end fashion is presented by the tool Stimulus by software company Argosim[1]. Stimulus lets the user define formalized requirements and enrich the system under development with state machines and block diagrams to include behavioral and architectural information, respectively. With the help of a build-in test suite, signals from the environment on which the systems depends and reacts can be simulated. The system behavior within these simulations is evaluated with regards to its constraints specified by the requirements and violations are detected. The main features include the detection of contradicting and missing requirements.

This tooling approach however exhibits some major differences to the methodology proposed in this paper. First and foremost, the form in which requirements are drafted in Stimulus is in a highly formalized manner from which this approach is to be differentiated. While there are many efforts within the research community to explicitly formalize requirements to improve on their validation possibilities [2], this markup language aims to provide the requirements engineer with a means to intuitively annotate natural language requirements in order to unfold the implicitly contained information in a way it can be used for validation purposes within a simulation. Secondly, the testing capability provided by Stimulus depends on the user to define inputs to the system and assign a range of values to them for test execution. This step however shall be automated with the proposed approach. From the data provided by the markups, a scenario for the simulation environment will be constructed, which evaluates the underlying constraints.

3 Approach

Our approach is schematically displayed in Figure 1. The starting point is a document of requirements formulated in natural language containing software specifications. The present requirements are written without pattern or other

[1] www.argosim.com

grammatical restrictions. With elements of our markup language the engineer marks key phrases. These are matched with signals of the simulation and system, which is called a mapping. The simulation is created automatically and the resulting scenario contains configurations of the simulation environment influenced by the selections made in the requirements. Simulation results are output as log files, which in connection with the mapping results, are fed back to the original requirements document. In addition to log data from traffic simulations, it is also possible to use real driver log data. In this way, real log data can be matched with natural language requirements. The simulation results or real data are displayed directly in the originally analyzed phrases.

In contrast to state of the art procedures, there is no necessity for translation into executable languages. Therefore, the entire scope of simulation options of the natural language requirements remains without any translation loss. Another improvement of today's standards lies in the testability of software at any state of development. First behaviors of the software can be simulated with simple markings early in the development process. Especially new assistance systems or functions such as autonomous driving are very complex and can only be tested with complex simulations. The test engineers therefore need a lightweight approach to evaluate requirements without formal translation.

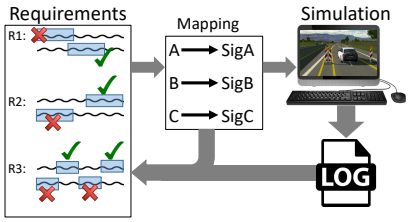

Fig. 1. Schematic representation of a requirements specification linked to a simulation with influencing intermediate steps

3.1 Markup Language

For marking software functions and environment conditions, we developed a lightweight multilevel markup language to connect requirements specifications and simulation runs. We developed our markup language to meet four demands.

First, a lightweight, intuitive approach for marking objects in natural language software requirements.

Second, a possibility to observe single objects as well as complex relations between elements in the simulation without a formal translation.

Third, an extraction of important simulation environment properties that must occur in the simulation.

Fourth, a possibility to evaluate software behavior already during the development process.

The resulting language consists of elements, which are assigned to phrases in the natural language requirements documents with defined content characteristics. This part of the process is performed by an engineer and is the starting point for the automated evaluation by the tool. Each element is assigned to one of four levels, which define the level of detail of the evaluation.

Elements: Elements are the basic component of our markup language. Available elements and their description are shown in Table 1. It also shows, how the elements are strictly associated with different levels of detail. The correct understanding of the elements by the engineer is crucial, since the manually performed labeling effects the type of automated simulation evaluation.

Levels: Figure 2 shows the four levels with the associated elements. The properties as well as the limits of the levels are explained in the following.

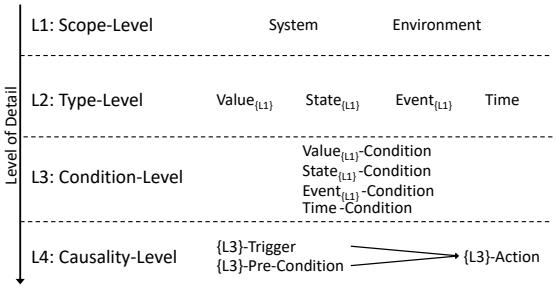

Fig. 2. Overview of levels and elements

The *Scope-Level* is used to differentiate between information on the system and on the simulation environment. As a result, the appearance of the objects in the simulation is displayed. However, no further information is available.

The *Type-Level* distinguishes the phrase of Level 1 into different types of text phrases depending on the behavior in the system. The different Level 2-types influence the type of evaluation and are the basis for the definition of conditions in Level 3.

The *Condition-Level* connects a type of Level 2 with a specific value via comparison operators to create condition statements. However, the formulated conditions have no connection among each other.

The *Causality-Level* establishes a relationship between the conditions of Level 3 and creates causal relationships. This requires detailed knowledge of the system and the necessary work process performed by the user is time consuming. The result however is an in-depth evaluation.

3.2 Marking Requirements

There are two main motivations to use this approach: to find information needed in order to choose a suitable simulation scenario; and to check or monitor

Table 1. Overview of all elements

Level	Element	Description
1	System	describes all information concerning the system, including any property perceptible from the outside as well as internal information. Result: link to signal available or not available.
	Environment	describes information on the simulation environment (e.g., weather) and simulation properties (e.g., simulation duration), and checks fulfillment of scenarios before a simulation run. Result: link to signal available or not available.
2	Value$_{\{L1\}}$	characterized by a value-continuous range and linked to system or environment. Result: progression over simulation time.
	State$_{\{L1\}}$	describes objects with multiple possible, but exclusive states (e.g., door - open/closed). Result: all appearing states.
	Event$_{\{L1\}}$	once or sporadically occurring object, often associated with signals. Result: number of appearances and average intermediate time.
	Time	concrete time specifications; automatically linked to simulation time. Result: not presented.
3	Value$_{\{L1\}}$-Condition	values of Level 2 linked by $<; \leq; =; >; \geq; \neq$ with a number or parameter. Result: duration of the fulfilled condition.
	State$_{\{L1\}}$-Condition	states of Level 2 linked by $=$ or $\neq$ with a possible state. Result: frequency, and duration in percent of the fulfilled condition.
	Event$_{\{L1\}}$-Condition	event from Level 2 with the values 1 or 0 for appearance and non-appearance. Result: number of appearances and average intermediate time.
	Time-Condition	time statements from Level 2 linked by $<; \leq; =; >; \geq$ or by natural language expressions such as "longer," "shorter," or "within"; must be linked to other conditions as an extension of other Level 3 conditions. Result: not presented.
4	{L3}-Trigger	Level 3 statements linked by AND,OR; if condition is fulfilled, {L3}-Action is triggered. Result: number of appearances.
	{L3}-Pre-Condition	Level 3 statements linked by AND,OR; pre-condition must be fulfilled in order to start a {L3}-Action. Result: number of appearances in total and as pre-condition with percentage.
	{L3}-Action	Level 3 statements linked together; following a {L3}-Trigger or {L3}-Pre-Condition. Result: number of appearance.

functionalities of a software component in different states of development. Our markup language facilitates the highlighting of necessary information and the observation in the simulation with an adaptable level of detail.

Figure 3 shows three example requirements [CL-1, CL2, CL-3] of a Cornering Light in a car, which is automatically switched on when turning. The dynamic and static cornering light function improves illumination of the road ahead when cornering.

Requirement Specification: Cornering Light

ID	Object Text
CL-1 L1/L2	At v < vMaxInd, the indicator ($State_S$) has priority over the steering wheel angle ($Value_S$) (e.g., roundabout (Environment))
CL-2 L2	Cornering light ($State_S$) is activated according to the active indicator
CL-3 L2	Cornering light ($State_S$) is deactivated at v ($Value_S$) > vMax

Fig. 3. Example of requirements of a Cornering Light System with initial marks of elements on Level 1 and 2

CL-1 contains Level 1 (Environment) and Level 2 ($State_S$ and $Value_S$) markings. If the aim is to ensure the presence of a roundabout in the simulation, this element will be marked with "Environment". The "State" mark for the indicator represents all occurring states. To observe the angle of the steering wheel and to evaluate the simulation duration, the choice of Level 2 mark "Value" is necessary.

CL-2 and CL-3 contain only Level 2 ($State_S$) markings. The subscript S stands for system; the subscript E describes an element of the simulation environment. Concerning objects of the environment, their occurrence is checked before runtime of the simulation scenario.

Figure 4 shows the identical requirements with an unchanged CL-1, but continually edited Level 2 objects in CL-2 and CL-3. By linking these objects to a newly marked condition, a Level 3 statement was created. A special feature is the link of a time condition in order to extend another condition. A time condition can exclusively be linked to other conditions.

In Figure 5, which is again showing the identical requirements, the Level 3 elements in CL-3 are brought into a relationship with each other by manually selecting them. By this causal relationship, Level 4 is reached.

Figure 5 displays the result of the development process performed by the engineer, in this example with appearance of all four levels. With these marked

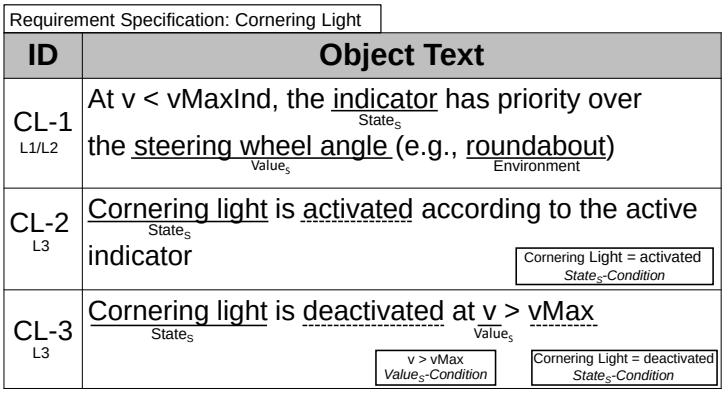

Fig. 4. Example of Requirements of a Cornering Light System with adopted marks with increasing complexity on Level 1, 2 and 3

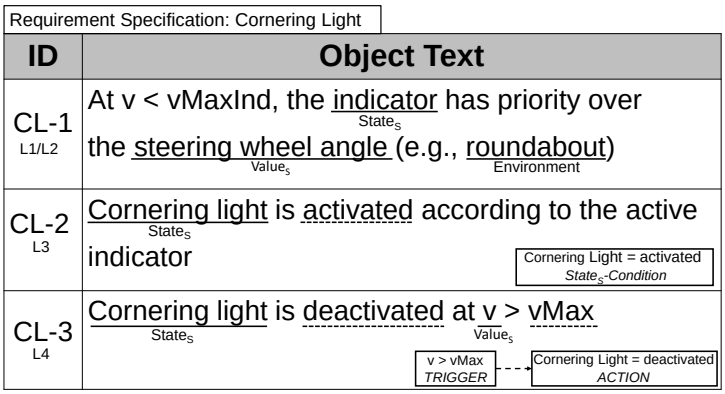

Fig. 5. Example of requirements of a Cornering Light System with complex marks on Level 1,2,3 and 4

requirements, simulations can now be carried out and evaluated in different depth of detail.

3.3 Simulation Execution and Representation

Before the start of the simulation, the marked text passages are mapped to signal names, like shown in Table 2. The signal names may be internal signals of the system or signals of the simulation environment. However, mapping is not always feasible if the matching signal does not exist in the simulation. This might indicate that the choice of scenario is not suitable or that the state of development is still too early. Nonetheless, it is still possible to start the simulation and just validate a subset of the system requirements. Further, the markups from the text and the signals from the simulation are not necessarily a one-to-one but

can also be established as a one-to-many mapping. The expressions "Cornering Light" and "indicator" in Table 2 demonstrate such a mapping. When tested in a simulation run, either of the two mapped signals can produce validation results for the annotated requirement since both should exhibit the same behavior according to their shared specification. After the preparation of the requirements

Table 2. Mapping from natural language expressions to signal names

signals	roundabout	→	<RB_Stat>
	vehicle speed	→	<V_vehicle>
	steering wheel angle	→	<Angle>
	Cornering light	→	<CL_Left_Stat>
			<CL_Right_Stat>
	indicator	→	<Indicator_Left_Stat>
			<Indicator_Right_Stat>
constants	vMax	→	55
	vMaxInd	→	30

document, the simulation can now be started. An excerpt of a possible resulting log file after running the simulation with CL-1, CL-2 and CL-3 is shown in Table 3.

Table 3. Excerpt of a log file

Vehicle ID	Source	Target	Signal	Value	Time
veh_1	centralBox	ECU	V_vehicle	40	88.010
veh_1	indicator	ECU	Indicator_Left_Stat	0	88.020
veh_1	cl	ECU	CL_Left_Stat	0	88.030
veh_1	centralBox	ECU	Angle	0	88.040
veh_1	centralBox	ECU	V_vehicle	25	89.010
veh_1	indicator	ECU	Indicator_Left_Stat	1	89.020
veh_1	cl	ECU	CL_Left_Stat	1	89.030
veh_1	centralBox	ECU	V_vehicle	20	90.010
veh_1	centralBox	ECU	Angle	10	90.040
veh_1	centralBox	ECU	V_vehicle	20	91.010
veh_1	centralBox	ECU	Angle	90	91.040
veh_1	centralBox	ECU	V_vehicle	25	92.010
veh_1	centralBox	ECU	Angle	10	92.040
veh_1	centralBox	ECU	V_vehicle	40	93.010
veh_1	indicator	ECU	Indicator_Left_Stat	0	93.020
veh_1	cl	ECU	CL_Left_Stat	0	93.030
veh_1	centralBox	ECU	Angle	0	93.040

The log data provided shows that the indicator is turned on at simulation time 89.020. The vehicle speed being 25 km/h fulfills the condition for the cornering light to be turned on, which the simulation log shows occurred at time

89.030. The values of the vehicle speed and steering angle then indicate that the vehicle made a turn. After the successful turn the indicator is turned off at time 93.020, which leads to the disabling of the cornering light. The table does not show the values of the indicator and the cornering light for simulation time 90 through 92, since no changes occurred during that time; with the indicator activated and the velocity under 30 km/h, the cornering light keeps being activated as intended.

This example of a small selection of requirements in a simple simulation scenario emphasizes the possible dimensions of a log file based on a whole requirements document with a comprehensive simulation. The extent also increases with the simulation duration.

An essential feedback mechanism is the presentation of the results in the original requirements document, depending on the chosen elements in the requirements and the analysis of the log data. Figure 6 shows the presentation of the evaluation results based on the simulation run in the presented example.

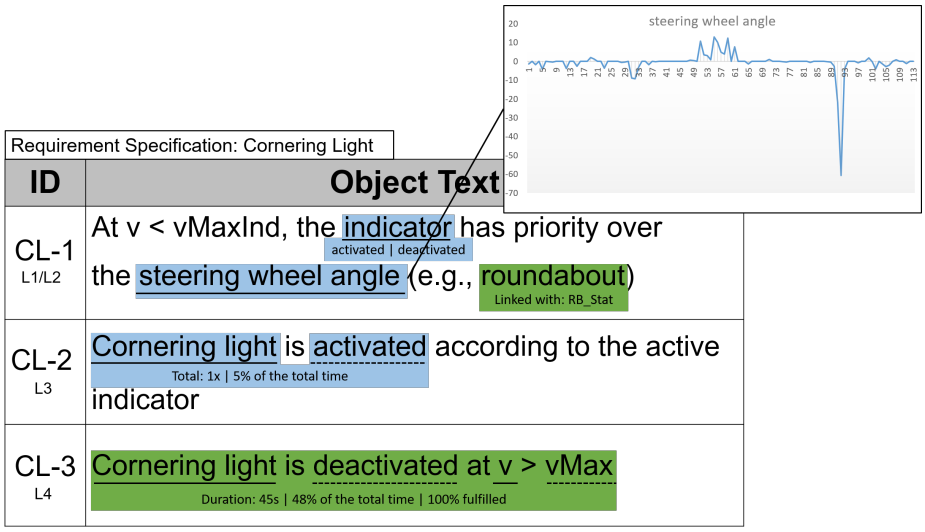

Fig. 6. Resulting marks of a simulation evaluation

In CL-1, the environment phrasing “roundabout” is mapped to the signal “RB_Stat”. The element “indicator” belongs to Level 2, therefore all occurring states can be displayed. In contrast, “steering wheel angle” is a value and an element of Level 2 and therefor it can be displayed graphically over the entire simulation time. If the steering wheel is turned to the left, the value is negative, so right turns are positive. Values below 20 degrees are lane changes. Large peaks between 45 and 90 degrees show the process of turning off.

In Cl-2, two conditions belonging to Level 3 are displayed. Depending on the selected condition, the according number of appearances is output. Additional in-

formation on $State_{\{L1\}}$-*Conditions* is the percentage of fulfillment over the simulation duration. For $Event_{\{L1\}}$-*Conditions*, information about average occurrence is available. The possibility of linking these conditions by *Time-Conditions* is not displayed; however the latter can not be used alone.

In CL-3, a *Trigger* and a dependent *Action* as elements of Level 4 are shown. Regarding the *Trigger*, information on total appearance is available. Further *Action*-related information is the number of appearances. In combination with the excerpt of the log file, the tool can confirm the causality of CL-3 and consequentially the given requirement as fulfilled.

The given example illustrates the influence of the specification degree on possible evaluation options. For basic analysis or early system development stages, lower and less time-consuming evaluation levels are suitable. However the tool also includes more complex options of evaluation. Though increasing complexity requires an increasing effort, evaluation and validation of entire requirements is possible.

4 Experiment

The approach was used in the automotive context to perform an experiment. Two aspects were examined: practicality of the language and identification of errors in its implementation. The following paragraph describes the structure, and execution. After that, results of the experiment as well as a summary of the advantages and disadvantages are subject of discussion.

4.1 Experimental Design

Object of the experiment are natural language requirements of Daimler AG. The used specifications describe the Intelligent Light System. Overall, the specification contains 3464 requirements and is divided into various subsystems. Among other things, it includes cross light, motorway light and cornering light. The requirements of the cornering light system used in Chapter 3 is extended by four further requirements and implemented in a separate vehicle function. All used requirements are part of an export of a DOORS database. They are written in natural language without limitation to patterns, or other structural or grammatical constraints. To carry out the marking process, the requirements are managed in a self-developed tool. The test engineer chooses a selection of marks according to the desired levels of results. In the presented experiment, the engineer marks a total of 13 text passages on all 4 levels, consisting of two markings on Level 1 and 2, five markings on Level 3 and four on Level 4.

Next step is linking the marked text passages to the signals of the simulation. This supports the creation of a scenario, which itself is the starting point for the simulation framework VSimRTI [11]. This framework links different simulators together and enables the virtual modeling of complex systems. Major simulator is the tool SUMO [6] developed by DLR, used for traffic simulation. VSimRTI makes it possible to equip SUMO vehicles with additional self-developed functions. An excerpt of an OpenStreetMap of Berlin is used for a realistic road

network and traffic light settings. Inclusion of all environment properties from Level 1 is checked before the start of the simulation run. The simulation run is performed 100 times with varying driving routes and an average of 53 vehicles involved. Each run is taking 163 seconds.

The vehicle function essentially consists of three parts: two cornering lights (left and right), and a central control unit. All components in the vehicle communicate via a virtual data bus, which is a modeled CAN communication. Each message sent and received via the bus is also written to a log file. This log file contains the time stamp, sender and recipient of the message. In addition, for each simulation step, vehicle data such as the steering angle, vehicle speed and status of the indicators are included in the log data. Following the simulation, the log data of all 100 vehicles is loaded by the developed tool. Evaluating these log data with the signal mapping from the initial step is closing the loop.

4.2 Results and Discussion

The tool reads the log data and provides the results, depending on the marks and the selected levels, in three categories: requirement fulfilled in green and not fulfilled in red (available for Level 1 and 4); and information available in blue (available for Level 2 and 3).

Figure 7 shows the presentation of the results in the original requirements for this experiment. Level 1, 2 and 3, as part of CL-1, CL-2 and CL-4, show the evaluation of word groups. One colored markup is displaying the results over the 100 simulation runs. Level 4 displays the evaluation of a causal connection between two items. For example in CL-3, $v > vMax$ was marked as trigger for the action *Cornering light* is *deactivated*. Since the requirement is fulfilled over the entire simulation time, it is colored green.

Even more complex are the requirements CL-5, CL-6 and CL-7, which also include Level 4 evaluations. Here, the triggers and actions have been marked and linked with each other across several requirements. The triggers *indicator is deactivated* (CL-5), $v < vMax$ (CL-5) and *curve radius* $>$ *angleOff* (CL-6) belong to the action *Cornering light is deactivated* in CL-6. A third Level 4 causality is also made up of the triggers from CL-5. In addition, the trigger *curve radius* $<$ *angleOff* (CL-7) is connected with all three triggers of CL-5 and the action *Cornering light is activated* in CL-7. The results are displayed in the requirement with the included action.

The major issue concerning all development approaches is the discrepancy between natural language requirements specifications and functionality of the software. The approach presented here bridges the gap between requirements on the one hand and simulative testing on the other. Particularly complex systems can be studied by the lightweight method at each stage of development. The mapping process is currently done manually and is time consuming for large software systems. However, if the presented approach is used parallel to the development, the mapping can also be maintained in parallel. New systems can build on previous mappings. Nevertheless, more research will be needed in an automated mapping process. Another challenge are changing software design

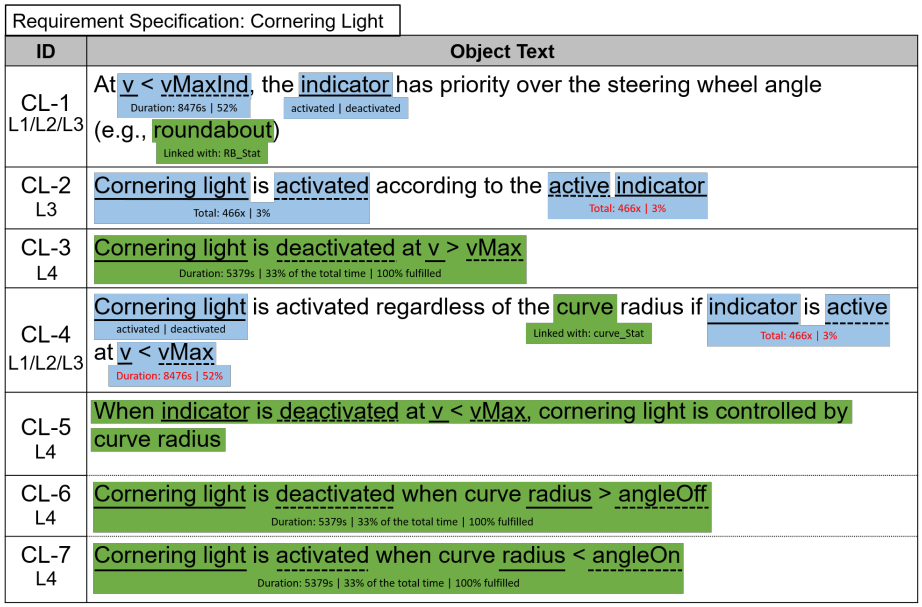

Fig. 7. Resulting marks of a simulation evaluation

decisions during the development process, which are not immediately updated in the original requirements documents. At present, this is performed at a later time, where some of the simulation and testing has already taken place. In our approach, the updates still have to be performed manually. However, the software engineer is motivated to perform the changes right away, so that the requirements documents always stay current and discrepancies during testing are prevented.

As the experiment shows, the evaluation can partly be sophisticated. This, however, is due to the complex requirements, which today are manually fragmented for testing. Our approach makes structuring and testing of conditions over multiple requirements possible.

5 Conclusion and Outlook

Complex software systems are based on ever larger requirement documents. Increasingly, these systems are being tested in simulations. State of the art is the translation of natural language requirements into executable models. Nowadays, the original requirements documents rarely influence the simulations and simulation results are usually not fed back to the specifications. Our lightweight multilevel markup language combines natural language requirements with simulations. Depending on the development stage, software functions can be monitored or complex requirements can be tested. The degree of detail of the evaluation can be determined by the tester. In our approach, there is no necessity

for translating original requirements documents. Our experiment with 100 simulation runs of a Cornering Light System, the processing procedure, and the evaluation report illustrates its usability and emphasizes the relevance to large and complex requirements documents.

In future, we plan further steps for automation. One possible approach can be the manual signal mapping automated through the use of machine learning. Another improvement might be automated identification of Level 2 states with natural language processing methods, which are subject of recent research. Due to the rapid increase of model-based development, markups in UML diagrams are also a focus of research.

References

1. Banks, J., Carson, J.S., Nelson, B.L., Nicol, D.M.: Discrete-Event System Simulation. Prentice Hall (2000)
2. Barmi, Z.A., Ebrahimi, A.H., Feldt, R.: Alignment of requirements specification and testing: A systematic mapping study. In: IEEE International Conference on Software Testing, Verification and Validation Workshops (2011). https://doi.org/10.1109/ICSTW.2011.58
3. Bjarnason, E., Runeson, P., Borg, M., Unterkalmsteiner, M., Engström, E., Regnell, B., Sabaliauskaite, G., Loconsole, A., Gorschek, T., Feldt, R.: Challenges and practices in aligning requirements with verification and validation: a case study of six companies. Empirical Software Engineering **19**(6) (2014). https://doi.org/10.1007/s10664-013-9263-y
4. Eckhardt, J., Vogelsang, A., Femmer, H., Mager, P.: Challenging incompleteness of performance requirements by sentence patterns. In: IEEE International Requirements Engineering Conference (RE) (2016)
5. Hasling, B., Goetz, H., Beetz, K.: Model based testing of system requirements using UML use case models. In: International Conference on Software Testing, Verification, and Validation (2008). https://doi.org/10.1109/ICST.2008.9
6. Krajzewicz, D., Bonert, M., Wagner, P.: The open source traffic simulation package sumo (06 2006)
7. Lucassen, G., Robeer, M., Dalpiaz, F., van der Werf, J.M.E.M., Brinkkemper, S.: Extracting conceptual models from user stories with visual narrator. Requirements Engineering **22**(3) (2017). https://doi.org/10.1007/s00766-017-0270-1
8. Mavin, A., Wilkinson, P., Harwood, A., Novak, M.: Easy approach to requirements syntax (ears). In: 2009 17th IEEE International Requirements Engineering Conference. pp. 317–322 (2009). https://doi.org/10.1109/RE.2009.9
9. Mich, L., Franch, M., Novi Inverardi, P.: Market research for requirements analysis using linguistic tools. Requirements Engineering **9**(2), 151–151 (2004)
10. Nebut, C., Fleurey, F., Traon, Y.L., Jezequel, J.M.: Automatic test generation: a use case driven approach. IEEE Transactions on Software Engineering **32**(3) (2006). https://doi.org/10.1109/TSE.2006.22
11. Schünemann, B.: V2x simulation runtime infrastructure vsimrti: An assessment tool to design smart traffic management systems. Computer Networks **55**(14), 3189 – 3198 (2011)